mips ejtag原理和实现

乔崇

June 16, 2014

Contents

Abstract

EJTAG是mips的onchip debug调试标准。现在龙芯1号、龙芯2号(龙芯2F和以前版本不支持)和龙芯3号系列都支持ejtag调试。通过ejtag可以大大方便软件调试,这里讲讲ejtag原理和ejtag-debug软件。
ejtag-debug是我编写的一个ejtag调试调试工具，支持读写寄存器、内存、反汇编、执行用户编写的小程序、gdb远程调试和脚本语言。

1 ejtag基础：jtag

ejtag和jtag的关系:
ejtag利用了jtag pin,通过jtag tap状态机器,访问jtag寄存器。通过jtag寄存器来控制处理器进入ejtag异常，在异常里面访问dmseg(0xff2000000-0xff2fffff0)来实现和pc主机进行交互。这一章的内容是讲jtag信号、状态机和寄存器。[1]

1.1 jtag信号

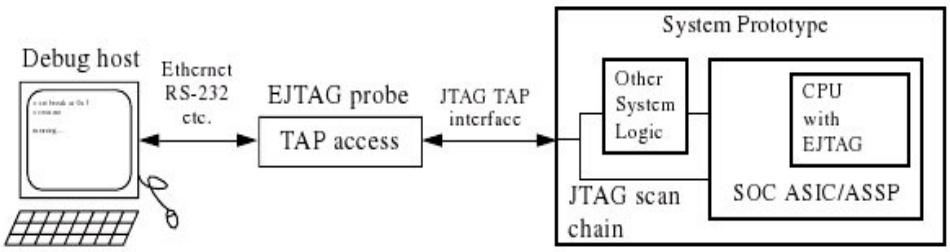

Figure 1: ejtag连接图

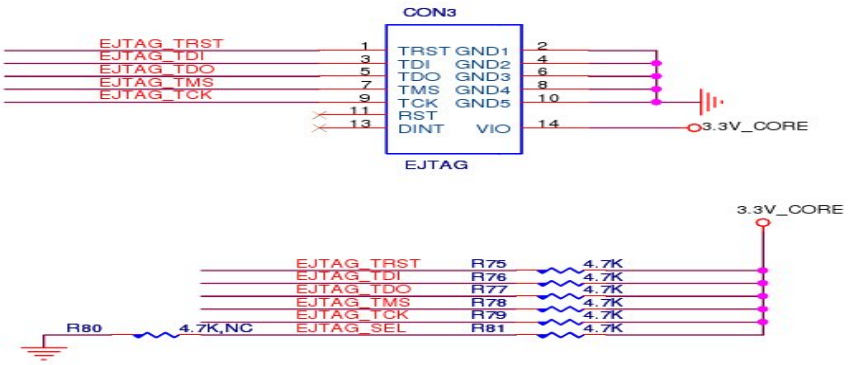

Figure 2: ejtag座信号定义

[1] jtag参考ieee1149.1,ejtag参考MD00047-2B-EJTAG-SPC-03.10.pdf

TRST	test reset 复位任选
TDI	test data in 串行输入到cpu
TDO	test data out 从cpu串行输出
TCLK	test clock 时钟
TMS	test mode select 状态机控制

在tclk的上升沿采集tms,tdi,tdo的信号。

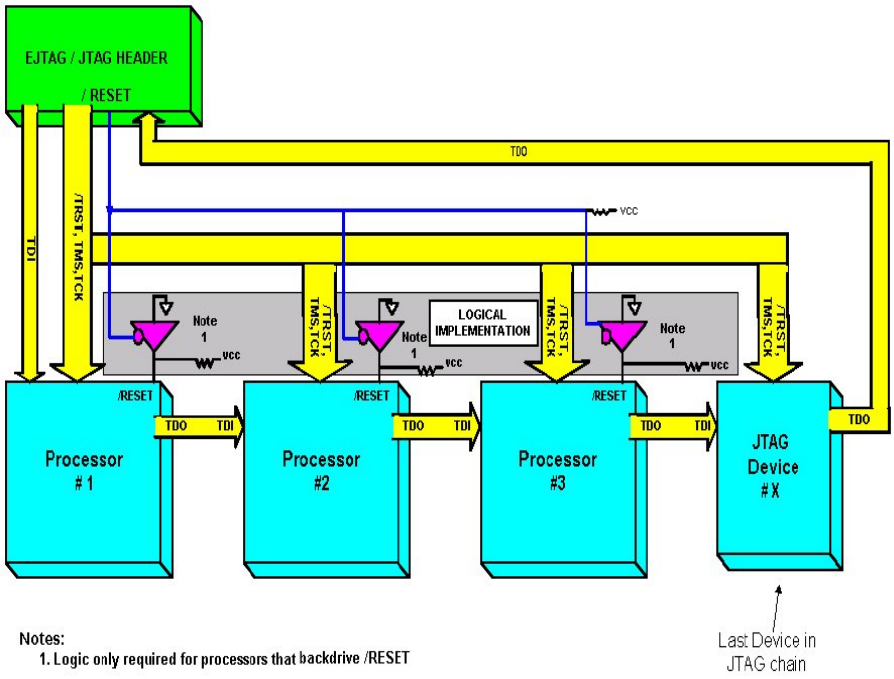

Figure 3: jtag chain

当有多个设备通过jtag连接起来的时候，将设备的tdo连接接到另一个设备的tdi上，成为一个jtag链接。
tms,tclk,trst信号是直接连到每个设备上的。

1.3 jtag状态机

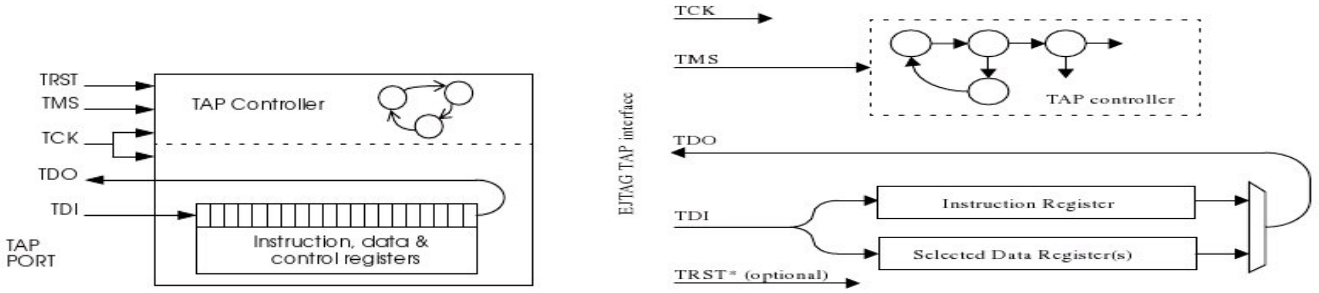

Figure 4: ejtag连接图

上面的两个图形象的画出ejtag tap controller工作的过程，一个是移位，另一个是ir,dr寄存器。通过状态机设置ir，访问dr，就实现访问ir指向的寄存器。ejtag ir是5位，dr是32-64位。

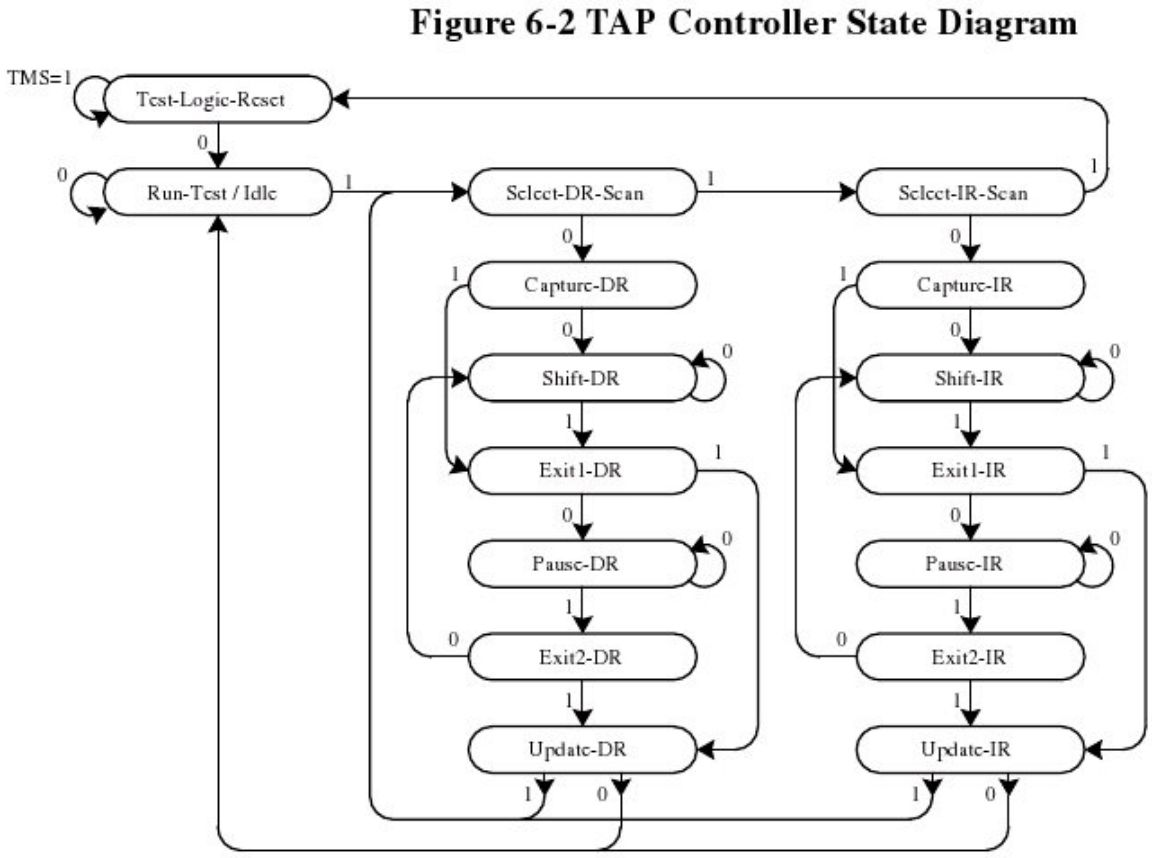

Figure 5: jtag状态机

通过jtag状态机读写ir,dr寄存器,来访问jtag/ejtag的寄存器。

1.4 jtag的寄存器

ejtag标准扩展了jtag寄存器，如增加了address,data,control寄存器等。要实现ejtag调试主要是反复访问这3个寄存器。

Table 1: Table 6-1 TAP Instruction Overview

Code	Instruction	Function
All 0’ s	(Free for other use)	Free for other use, such as JTAG boundary scan
0x01	IDCODE	Selects Device Identi?cation (ID) register
0x02	(Free for other use)	Free for other use, such as JTAG boundary scan
0x03	IMPCODE	Selects Implementation register
0x04 - 0x07	(Free for other use)	Free for other use, such as JTAG boundary scan
0x08	ADDRESS	Selects Address register
0x09	DATA	Selects Data register
0x0A	CONTROL	Selects EJTAG Control register
0x0B	ALL	Selects the Address, Data and EJTAG Control registers
0x0C	EJTAGBOOT	Makes the processor take a debug exception after reset
0x0D	NORMALBOOT	Makes the processor execute the reset handler after reset
0x0E	FASTDATA	Selects the Data and Fastdata registers
0x0F	(EJTAG reserved)	Reserved for future EJTAG use
0x10	TCBCONTROLA	Selects the control register TCBTraceControl in the Trace Control Block
0x11	TCBCONTROLB	Selects another trace control block register,Used to access the registers specified by the TCBCONTROLBREG field
0x12	TCBDATA	and transfers data between the TAP and the TCB control register
0x13	TCBCONTROLC	Selects another trace control block register
0x14	PCSAMPLE	Selects the PCsample register
0x15 - 0x1B	(EJTAG reserved)	Reserved for future EJTAG use
0x1C - All 1’ s	(Free for other use)	Free for other use, such as JTAG boundary scan
All 1’ s	BYPASS	Select Bypass register

1.5 访问jtag寄存器代码例子

下面代码是通过并口控制jtag信号，访问jtag寄存器。

```
//
// fJTAG_Wiggle
//
// Sets states of lines out to JTAG controller and returns
// the one input bit from status (which is inverted in hardware)
//
unsigned char fJTAG_Wiggle(unsigned char uc_tms, unsigned char uc_tdo)
{
  unsigned char uc_dout;
  unsigned char uc_din;

  uc_dout = (unsigned char)(mPARPORT_TRS_BIT(1)|mPARPORT_TMS_BIT(uc_tms)|
                            mPARPORT_TDI_BIT(uc_tdo)|mPARPORT_TCK_BIT(0)|
                            mPARPORT_RST_BIT(1));
  fPARPORT_Write(uc_dout);

  uc_dout = (unsigned char)(mPARPORT_TRS_BIT(1)|mPARPORT_TMS_BIT(uc_tms)|
                            mPARPORT_TDI_BIT(uc_tdo)|mPARPORT_TCK_BIT(1)|
                            mPARPORT_RST_BIT(1));
  fPARPORT_Write(uc_dout);

  // Read in the status bit (inverted in hardware)
  uc_din = fPARPORT_Read() & dPARPORT_TAP_DO ? 0 : 1;

  return uc_din;
}
```

```
27
28 //
29 // fJTAG_Instruction
30 //
31 // Read/Write from/to the instruction register
32 //
33 unsigned char fJTAG_Instruction(unsigned char uc_data_out)
34 {
35   unsigned char uc_data_in;
36   unsigned char uc_result;
37   unsigned char uc_index;
38   unsigned char iri,iro;
39   int cpu;
40
41   last_ir = uc_data_out;
42
43   fJTAG_Wiggle(1, 0); // enter select-dr-scan
44   fJTAG_Wiggle(1, 0); // enter select-ir-scan
45   fJTAG_Wiggle(0, 0); // enter capture-ir
46   fJTAG_Wiggle(0, 0); // enter shift-ir (dummy)
47
48   iri = uc_data_out;
49
50   // Clock all but last bit out
51   for(cpu=core_cpucount-1;cpu>=0;cpu--)
52   {
53    uc_data_in=0;
54    uc_data_out=(cpu==core_cpuno)?iri:0x1f;
55   for (uc_index=0;uc_index<5; uc_index++)
56     {
57       uc_result   = fJTAG_Wiggle((cpu==0&&uc_index==4)?1:0, (uc_data_out & 0x01));
58       uc_data_out = uc_data_out >> 1;
59       uc_data_in = uc_data_in | (uc_result << uc_index);
60     }
61     if(cpu==core_cpuno)iro=uc_data_in;
62   }
63
64   fJTAG_Wiggle(1, 0); // enter update-ir
65   fJTAG_Wiggle(0, 0); // enter runtest-idle
66
67   return iro;
68 }
```

2 ejtag实现

1. ejtag标准里面提到的ejtag最小实现包括:ejtag特殊指令(sdbbp,deret)，ejtag异常。
   在这个最小实现里面软件可以通过sdbbp指令触发ejtag异常(rom中)，ejtag异常用开管理断点等，deret退出异常。

2. 要通过jtag进行调试的最小实现:ejtag特殊指令(sdbbp,deret)，ejtag异常,tap controller,dmseg。
   通过写tap控制器debug寄存器触发异常，通过访问dmseg和主机实现通信。

3. 1,2要实现硬件断点必须实现drseg。

2.1 ejtag特殊指令

sdbbp: 指令实现软件断点功能，当处理器执行这条指令时进入ejtag异常。
deret: 执行实现从ejtag异常退出。

2.2 ejtag地址空间

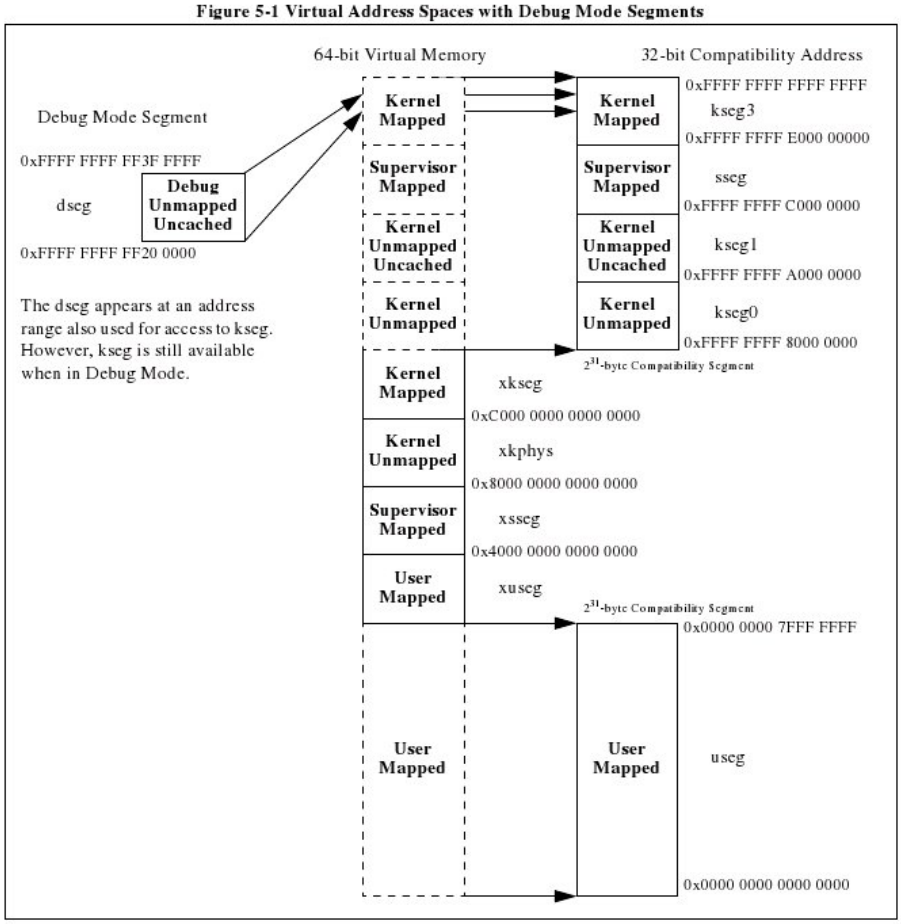

Figure 6: ejtag地址空间

ejtag地址空间包含dmseg,drseg.
The dseg segment is subdivided into dmseg (EJTAG memory) segment and the drseg (EJTAG registers) segment. The dmseg segment is used when the probe services the memory segment. The drseg segment is used when the memory-mapped debug registers are accessed. Table 5-2 shows the subdivision and attributes for the segments.

Table 2: Table 5-2 Physical Address and Cache Attribute for dseg, dmseg and drseg

Subsegment Name	Virtual Address	Cache Attribute
dmseg	0xFFFF FFFF FF20 0000 to 0xFFFF FFFF FF2F FFFF	uncached
drseg	0xFFFF FFFF FF30 0000 to 0xFFFF FFFF FF3F FFFF	uncached

Because the dseg segment is serviced exclusively by the EJTAG features, there are no physical address per se. Instead the lower 21 bits of the virtual address select the appropriate reference in either EJTAG Uncached memory or registers. References are not mapped through the TLB, nor do the accesses appear on the pexternal system memory interface.
cpu可以从dmseg中取指令执行，指令里面可以访问drseg来设置硬件中断等。

2.3 ejtag异常

Table 3: The same debug exception vector location is used for all debug exceptions. The ProbTrap bit in the EJTAG Control Register (ECR) in the optional Test Access Port (TAP) determines the vector location.

ProbTrap bit in ECR register	Debug Exception Vector Address
0	0xFFFF FFFF BFC0 0480
1	0xFFFF FFFF FF20 0200 in dmseg

3a开始的时候不能从ejtag取地址,就是从rom取址的。

2.3.1 从rom取地址过程

当ejtag进入异常处理程序的时候跳到0xffffffffbfc00480执行，我们可以在异常处理程序中发出对dmseg的访问，这样就是实现了cpu通过ejtag和主机进行通信。

1. pc机端，通过并口读ejtag的控制寄存器(ir=10)，注意读的同时会串入一个新值，看读回来的DM(bit3)是否为1,为1则在debug模式,goto 3,否则 goto 2
2. 写control reg(ir=10),设置protrap(bit14)0,break为(bit12)1，使得进入debug模式，且从rom取址，goto 1
3. 看debug寄存器的pracc(bit 18)是否为1，为则表明cpu想ejtag dmseg地址发出访问,debug寄存器的Prnw(bit19)表明是读还是写，psz(bit 30:29)表明访问宽度，goto 4,否则goto 1
4. 读ejtag的address reg(ir=8),如果是写操作goto 5,否则goto 6
5. 读ejtag的data reg(ir=9),读出写过来的内容,goto 7
6. 将数据写入ejtag data reg(ir=10),goto 7
7. 写debug寄存器，清掉pracc位，这样cpu这次访存操作就完成了。

2.3.2 从ejtag dmseg取地址过程

当ejtag进入异常处理程序的时候跳到0xffffffffff200200执行，pc主机通过ejtag发送指令回去，控制cpu执行，最后运行deret退出ejtag debug模式。

读内存一个字节的程序

```
#define FIFO     0x1F8
#define STACK    0x1F0
#define PARAM    0x1E8
#define tgt_exec(s,len) ({unsigned int codebuf[]=tgt_compile(s);fMIPS32_ExecuteDebugModule(codebuf,min(
    len,sizeof(codebuf)),1);})
unsigned char jtag_inb(long long addr)
```

```
{
    ui_param[ui_param_i++]=addr;
    tgt_exec(\
    ".set noat;\n" \
    "1:\n" \
    "mtc0   $15, $31;\n" \
    "li    $15,0xFF200000;\n" \
    "sd    $2, STACK($15);\n" \
    "sd    $3, STACK($15);\n" \
    "ld    $2, PARAM($15);\n" \
    "lb    $3, ($2);\n" \
    "sw    $3, PARAM($15);\n" \
    "ld    $3, STACK($15);\n" \
    "ld    $2, STACK($15);\n" \
    "mfc0   $15, $31;\n" \
    "b 1b;\n" \
    "nop;\n" \
    ,0x80000);
    return (unsigned char)(ui_param[--ui_param_o]);
}
```

1. pc机端，通过并口读ejtag的控制寄存器(ir=10)，注意读的同时会串入一个新值，看读回来的DM(bit3)是否为1,为1则在debug模式,goto 3,否则 goto 2
2. 写control reg(ir=10),设置protrap(bit14)1,break为(bit12) 1，使得进入debug模式，且从dmseg取址，goto 1
3. 看debug寄存器的pracc(bit 18)是否为1，为则表明cpu想ejtag dmseg地址发出访问,debug寄存器的Prnw(bit19)表明是读还是写，psz(bit 30:29)表明访问宽度，goto 4,否则goto 1
4. 读ejtag的address reg(ir=8)到变量addr,如果是写操作goto 5,否则goto 6
5. 读ejtag的data reg(ir=9),读出写过来的内容,如果地址大于0xff200200,(addr-0xff200200)/4是codebuf偏移offs,则codebuf[offs]=data,如果地址小于0xff200200,param,stack=data等,goto 7
6. 如果地址大于0xff200200,(addr-0xff200200)/4是codebuf偏移offs,则data=codebuf[offs],如果地址小于0xff200200,data=param,stack等,将data写入ejtag data reg(ir=10),goto 7
7. 写debug寄存器，清掉pracc位，这样cpu这次访存操作就完成了。

2.4 ejtag特殊cp0寄存器

cp0_debug: cp0 $23
cp0_depc: cp0 $24
cp0_desave: cp0 $31 当产生ejtag debug异常的时候,cpu将当前的pc值保存到cp0_depc中，从ejtag exception返回的时候edert将cp0_depc.
读cp0_debug可以判断出ejtag中断类型.cp0_desave是给软件保存寄存器用。

检查ejtag状态代码

```
int check_ejtag(char *buf,int *debug_reg)
{
    long long ui_result,debug_reg1;
    char *p = buf;
    *p=0;

    ui_result = dMIPS32_ECR_CPU_RESET_OCCURED | dMIPS32_ECR_ACCESS_PENDING |
        dMIPS32_ECR_PROBE_ENABLE | dMIPS32_ECR_PROBE_VECTOR;

    fJTAG_Instruction(dJTAG_REGISTER_CONTROL);
    ui_result = fJTAG_Data(ui_result);

    // Check for debug mode
    if (ui_result & dMIPS32_ECR_IN_DEBUG_MODE)
    {
        long long cur_addr;
```

```
        int  cur_ins;

        // Read out the debug register to determine the cause
        static int codebuf[100] =tgt_compile(
                "1:\n"
                "mtc0   ra, $31;\n"
                "lui ra,0xff20;\n"
                "sd v0, STACK(ra);\n"
                "mfc0   v0,  $23;\n"
                "sd v0, PARAM(ra);\n"
                "mfc0   v0, $24;\n"
                "sd v0, PARAM(ra);\n"
                "mfc0   v0, $24;\n"
                "lw     v0,(v0)\n"
                "sd v0, PARAM(ra);\n"
                "ld v0, STACK(ra);\n"
                "mfc0   ra, $31 ;\n"
                "b 1b;\n"
                "nop;\n"
                );

        fMIPS32_ExecuteDebugModule(codebuf,0x8000,1);
        cur_ins = ui_param[--ui_param_o];
        cur_addr = ui_param[--ui_param_o];
        debug_reg1 = ui_param[--ui_param_o];

        p += sprintf(p,"cpu %d debug:%08x #",core_cpuno,debug_reg1);

        if (debug_reg1 & dMIPS32_DEBUG_SW_BKPT)
            p += sprintf(p,"i");

        if (debug_reg1 & dMIPS32_DEBUG_HW_INST_BKPT)
        {
            p += sprintf(p,"I ");
        /*clear inst breakpoint*/
            tgt_exec(
                    "1:\n"
                    "mtc0    $15, $31;\n"
                    "lui    $15,0xFF30;\n"
                    "sw $0,0x1000($15);\n"
                    "mfc0    $15, $31;\n"
                    "b 1b;\n" \
                    "nop;\n" \
                    ,0x80000);
        }

        if (debug_reg1 & dMIPS32_DEBUG_HW_DATA_LD_BKPT)
            p += sprintf(p,"l");

        if (debug_reg1 & dMIPS32_DEBUG_HW_DATA_ST_BKPT)
            p += sprintf(p,"s");

        if (debug_reg1 & dMIPS32_DEBUG_HW_DATA_LD_IMP_BKPT)
            p += sprintf(p,"L");

        if (debug_reg1 & dMIPS32_DEBUG_HW_DATA_ST_IMP_BKPT)
            p += sprintf(p,"S");

        if(debug_reg1 & (dMIPS32_DEBUG_HW_DATA_LD_BKPT|dMIPS32_DEBUG_HW_DATA_ST_BKPT|
             dMIPS32_DEBUG_HW_DATA_LD_IMP_BKPT|dMIPS32_DEBUG_HW_DATA_ST_IMP_BKPT))
        {
        /*clear data breakpoint*/
            tgt_exec(
                    "1:\n"
                    "mtc0    $15, $31;\n"
                    "lui    $15,0xFF30;\n"
```

```
                    "sw $0,0x2000($15);\n"
                    "mfc0   $15, $31;\n"
                    "b 1b;\n" \
                    "nop;\n" \
                    ,0x80000);
        }

        if (debug_reg1 & dMIPS32_DEBUG_SINGLE_STEP_BKPT)
            p += sprintf(p," step");

        if (debug_reg1 & dMIPS32_DEBUG_JTAG_BKPT)
            p += sprintf(p,"user ");
        if (debug_reg1 & 0x100000)
            p += sprintf(p," exp");

        p += sprintf(p,"\n");
        md_disasm (p, cur_addr, cur_ins);
    *debug_reg = debug_reg1;

    }
    else
    {
        p += sprintf(p,"cpu %d not in debug mode",core_cpuno);
    *debug_reg = 0;
    }


    return ui_result;
}
```

2.5 ejtag的断点

软件断点:sdbbp指令
  需要修改代码，保存原指令，写入sdbbp,刷cache
硬件断点:ejtag数据空间
数据断点:

Table 4: Data Breakpoint Register Mapping

Offset in drseg	Register Mnemonic	Register Name and Description
0x2000	DBS	Data Breakpoint Status
0x2100 + 0x100 * n	DBAn	Data Breakpoint Address n
0x2108 + 0x100 * n	DBMn	Data Breakpoint Address Mask n
0x2110 + 0x100 * n	DBASIDn	Data Breakpoint ASID n
0x2118 + 0x100 * n	DBCn	Data Breakpoint Control n
0x2120 + 0x100 * n	DBVn	Data Breakpoint Value n

  指令断点:

Table 5: Instruction Breakpoint Register Mapping

Offset in drseg	Register Mnemonic	Register Name and Description
0x1000	IBS	Instruction Breakpoint Status
0x1100 + 0x100 * n	IBAn	Instruction Breakpoint Address n
0x1108 + 0x100 * n	IBMn	Instruction Breakpoint Address Mask n
0x1110 + 0x100 * n	IBASIDn	Instruction Breakpoint ASID n
0x1118 + 0x100 * n	IBCn	Instruction Breakpoint Control n

3 ejtag-debug的架构

由很多小命令组成，支持脚本语言。

3.1 编译

```
git clone 10.2.5.27:/home2/qiaochong/ejtag-debug
cd ejtag-debug
make
```

默认的交叉工具链是mipsel-linux-gcc
如果不是需要
make CROSS_COMPILE=mipsel-linux-
make的时候采用mycc,mycpp.pl来实现将c先进行一次预编译处理将tgt_exec, tgt_compile转换成二进制数组。
编译生成ejtag_debug_pp和ejtag_debug_usb
ejtag_debug_pp是用在并口电缆
ejtag_debug_usb是用usb电缆

接着编译bin目录下的程序:
bin目录下的内容是一些小程序用来烧flash,md5sum等.
bin目录下的程序可以编译成elf32或者elf64格式的。

```
cd bin
make CROSS_COMPILE=mipsel-linux- test.elf
make CROSS_COMPILE=mipsel-linux- test.bin
make CROSS_COMPILE=mipsel-linux- test.elf64
make CROSS_COMPILE=mipsel-linux- test.bin64
```

test.bin是abi o32,test.bin64是abi n64.

3.2 运行方法

需要超级用户权限
sudo ./ejtag_debug_usb
配置文件是ejtag.cfg
程序会自动打开./ejtag.cfg并执行里面的内容。

3.3 ejtag-debug参数

./ejtag_debug_usb [-dlStch] [-e cmd] [-T n]
-d: verbose on,show debug messags
-e 'cmd': run cmd
-l: do not use read line
-S: log disassemble info
-s: run cmdserver
-t: disable timer
-T n: set timer n ms
-c: do not load cfg file
-h: show this help

3.4 ejtag-debug里面的目录结构

Table 6: ejtag-debug里的目录结构

目录/文件名	类型	功能
ejtag_debug_usb.exe	文件	usb ejtag的可执行文件
ejtag_debug_pp.exe	文件	并口ejtag的可执行文件
ejtag.cfg	文件	ejtag的配置文件，可执行行文件启动时候会先执行这个文件里面的内容
ejtag.rc	文件	在ejtag命令行上执行过的命令被保存在ejtag.rc中
Makefile,example.c,ejtag.a,include,mycc,mycpp	文件	ejtag 支持二次开发，example.c是一个事例程序
ddr.txt	文件	龙芯1b的ddr参数文件
mipsel-gdb	文件	mips gdb, gdb命令需要用到这个文件
scripts	目录	命令里面用到的脚本
configs	目录	ddr初始化和flash烧写脚本
bin	目录	bin文件开发环境，可以通过callbin执行
logic	目录	最新版本逻辑编程文件
driver	目录	windows下的驱动文件

3.5 ejtag-debug的命令

ejtag-debug由很多小命令组成，支持名字自动补齐。在行首加#表示注释。

- h [cmd] 查看帮助
- setconfig [configname] [value]
  重要的设置:
  helpaddr: 帮助地址，帮助程序执行的地址，地址应该指向ddr,cache/uncache都可以 put,get,callbin,call等命令会将帮助程序先上载到helpaddr上，然后执行helpaddr
  usb_ejtag.put_speed: 设置usb上传速度，16bit,越大越慢
  putelf.uncached: 设置putelf为uncached,1在put前刷cache,2不刷cache.
  core.cpucount : 设置cpu数目
  core.cpuno : 设置当前调试的cpu号
  core.cpuwidth : 设置cpu的数据宽度
  display.cmd : 当ejtag状态变化时候运行命令，如断点发生时。 jtag.showins : 当jtag状态变化的时候显示指令。 还有很到配置用setconfig命令可以列出来
- set [regname|regno] [value] 读写寄存器

```
    set 读出所有寄存器
    set pc 读pc
    set at 读at
    set pc 0xffffffffbfc00000
#设置pc的数值是0xffffffffbfc00000
```

- setenv [envname] [value] 设置环境变量
  ejtag向内核传参环境变量是通过ENV_开头的环境变量来设置的，如:
  setenv ENV_memsize 256
  setenv ENV_highmemsize 0
  setenv ENV_cpuclock 266000000
  分别是设置内核内存大小和cpu的时钟。
- d1-d8,d1q-d8q,m1-m8 1-8字节dump/modify的意思，实现的时候是函数指针,具体功能有先进行选择，默认是读内存。 相关的命令有mems,cp0s,regs,jtagregs,spiroms
- mems:select access mem
- cp0s:select access cp0
  读cp0_config1:

```
    cp0s 1
    d8 16 1
```

也可以这样用

```
cp0s 1 d8 16 1
```

- source file 从file文件中读取命令执行
- loop count cmd args.. 循环执行count此cmd args...
- echo args 打印args
- echo_on 回显示执行的命令
- echo_off 不回显执行的命令
- verbose [on|off|&fdno|filename] 打开或关闭debug调试，或存到fd中。
  下面命令实现打印出ejtag上执行的汇编代码

```
timer 0
verbose on
setconfig log.level 15
setconfug log.disas 1
```

- waitreg reg data 等待ejtag reg为data
- waitfacc [startaddr] [endaddr] 等代cpu访问dmseg且地址在startaddr-endaddr间或者为0xff200200
- goback [addr] 跳到dmseg addr
- run binfile 运行bin文件，如run gzrom.bin
- msleep n 等待n ms
- shell cmd 执行shell命令，或者调用脚本程序
- timer n nms检查一次ejtag状态,n==0的时候不检查
- b addr,设置软件断点，在addr上插入sdbbp
- unb addr 删除软件断点,恢复原来的指令
- hb addr [ibm] 设在硬件断点,ibm为1的位不比较地址

```
hb 0xbfc00000 0x7ffff #程序执行到0xbfc00000 0xbfc7ffff都会产生硬件断点
```

- unhb addr 删除硬件断点
- hbls 列出硬件指令断点
- watchls 列出硬件数据断点
- s单步执行
- uns取消单步执行
- cont 退出ejtag模式，继续执行
- cpu cpuno 切换cpu到cpuno
- map filemame start [size] 映射文件到dmseg start大小为size
- unmap filemame start [size] 取消映射文件到dmseg start大小为size，dmseg内容写代文件中
- memsetx/fmemsetx/smemsetx addr value size 在处理器上运行memset,其中x为1,2,4。1为字节写，
  2为双字写,4为4字节写。
- memcpy/fmemcpy/smemcpy saddr dassr size 在处理器上运行memcpy
- put/fput/sput file address ,上传文件到板子上,相关的配置usb_ejtag.put_speed,put.pack_size,put.md5sum
- get/fget/sget filename address size,从板子上下载文件,相关配置get.pack_size

- putelf/fputelf/sputelf elffile, 上传elf文件
- initrd initrdfile ,设置initrd
- karg "kernel args" ,设置内存参数
- disas address [count] ,反汇编
- callbin/fcallbin/scallbin binfile [arg0 arg1 ...],执行bin文件
- call/fcall/scall address [arg0 arg1 ...],调用函数
- erase 擦除整个flash芯片
- erase_area start end sectorsize 檫除部分flash芯片
- program ramaddr flashaddr size flash编程
  这些命令用来烧写flash，使用方法如下:(以后准备用callbin直接调用c语言来烧写flash)

```
setconfig flash.type st25vf080
erase_area 0 0x7ffff 0x10000
program 0x81000000 0 0x70000
```

- server [port] 启动命令server,另外一个ejtag-debug程序可以通过网络发命令过来执行,默认端口为8880
- shell program [args] 启动命令server同时，运行shell命令program
- gdbserver [port] 启动gdbserver,可以远程gdb remote链接调试,默认端口为50010
- gdb elffile 一个命令运行perl scripts/gdb.pl,实现后台启动gdbserver,前台运行gdb来调试elffile
- expr [expression|#regname]
  expr命令执行简单的+-*/\和进制转换，从左向右计算，支持括号.
  expr能读寄存器内容，主要为了方便调试用的
  翻引号里的内容会传给expr,并用结果替代，如
  disas `pc` 10 #反汇编pc开始10条指令
- newcmd cmdname oldcmd note,增加一个新的命令
- cache op addr size 发cache op从addr到addr+size
- cacheflush addr size 刷cache使得内存和cache一致，根据cache配置刷icache和dcache或者二级cache
- cache_config perl脚本读cpu的prid和config寄存器来设置cache参数，cache，cacheflush会用到这些参数
- selectcore coreno 龙芯3b选择调试的cpu,0xf是8个cpu串起来

3.5.1 callbin的实现

callbin调用的bin文件是pic位置无关代码，通过got段进行重定位。c语言的入口函数是mymain，callbin的参数直接作为mymain的参数。如果参数是字符串则callbin会将字符串地址拷贝到内存里面，并将内存地址作为参数。callbin中包含叫做tinyc的c库,实现printf,内存，字符串操作等。

实现memset的bin程序test.c

```
int mymain(char *buf,int len)
{
    printf("this is a test\n");
    memset(buf,c,len);
    return 0;
}
```

编译:

```
make CROSS_COMPILE=mipsel-linux- test.bin
```

调用:

```
callbin bin/test.bin 0xa0000000 0x12
```

默认打印是通过ejtag打印到pc机端，如果要打印到串口，可以定义putchar函数:

实现memset的bin程序test.c

```
int putchar(c)
{
 *(volatile char *)0xb4000000=c;
}
int mymain(char *buf,int len)
{
    printf("this is a test\n");
    memset(buf,c,len);
    return 0;
}
```

callbin的从定位和clear bss在start.S和start64.S中做的。got section存的是函数的地址，要用当前的的实际地址修正函数的地址和gp的值，然后才能正确执行。编译的时候的起始地址是0，运行的时候的起始地址是helpaddr。callbin源码位于lib/callbin.c中。

3.6 ejtag-debug的脚本语言

3.6.1 source命令

- ejtag-debug的source file和Source file命令可以file文件里面的命令依次执行.
- loop命令可以实现简单的固定循环
- expr命令可以作简单计算和寄存器值作为参数

```
d4 0xbfc00000+0x10 10
loop 10 d4 0xbfc00000 10
```

3.6.2 外部perl脚本

本来想在内部直接集成一种脚本语言，如awk，但后来发现比较麻烦。因此采用了shell命令方法来实现脚本语言。跟busybox原理差不多. 因为ejtag-debug刚启动的时候要打开设备作初始化，如果执行一个命令打开关闭一遍设备会性能很差。这里采用server,clint的方法来解决。要执行脚本语言前现启动server,会打开端口8880来接收命令。在另一个终端象这样运行

```
    ejtag-debug set
```

则会telnet 8880，输入命令，打印输出也重定向到网络。 上面的过程可以简化成1个shell命令

```
    shell ejtag-debug set
    shell perl scipts/test.pl
```

perl脚本分析
scripts/io.pm是将命令封装成perl的函数

- inb就是调用d1q
- outb就是调用m1
- inb/inh/inw/inq
- outb/outh/outw/outq
- do_cmd执行命令,直接输出到终端

- do_cmd1执行命令,结果返回

要写一个perl脚本test.pl访问ejtag，只要写一个perl程序，前面包含

```
#!/usr/bin/perl
use bignum;
push @INC,qq(./scripts);
require qq(io.pm);
```

后面就可以调用inb/outb/do_cmd等函数访问ejtag了。 调用的时候，运行:

```
    shell perl scipts/test.pl
```

test.pl

```
#!/usr/bin/perl
use bignum;
push @INC,qq(./scripts);
require qq(io.pm);
outb(0xffffffffbfe001e3,0x80);
outb(0xffffffffbfe001e0,0xd);
outb(0xffffffffbfe001e3,0x3);
printf "value is %x",inb(0xffffffffbfe001e3);
do_cmd("cont");
```

3.7 ejtag和gdb

为了实现通过通过gdb利用ejtag来调试程序,ejtag-debug中增加了gdbserver命令，可以直接执行gdb命令，内部调用gdbserver

3.7.1 gdb remote协议

运行info gdb命令可以查到相信的gdb remote标准。

```
    * Remote Protocol::          GDB Remote Serial Protocol
    * Packets:
```

运行gdb进行调试的时候，可以打开remote调试功能，能够打印出remote packet的内容

```
gdb) set debug remote -1
```

3.7.2 gdbserver的实现

- 打开tcp端口50010,设置成非阻塞方式,等待端口上的数据
- 解析packet,根据packet内容进行内存读写，设置断点等
- 查询ejtag状态,如过从正常状态进入ejtag状态，发送信号给gdb端

3.7.3 gdbserver的使用

运行gdb命令:

```
cpu0 -gdb /mnt/sdd3/work/3afirewall/linux-loongson-release/vmlinux
+ EJTAGEXE=/mnt/sdd2/work/bioscfg/ejtag_debug_pp
++ /mnt/sdd2/work/bioscfg/ejtag_debug_pp setconfig core.abisize
+ abisize=00000040
+ /mnt/sdd2/work/bioscfg/ejtag_debug_pp gdbserver 50010 1
++ mktemp /tmp/gdbXXX
+ tmpfile=/tmp/gdbDeq
+ '[' 00000040 = 00000040 ']'
+ echo 'set mips abi n64 '
+ echo 'target remote :50010'
+ mipsel-gdb -q -x /tmp/gdbDeq /mnt/sdd3/work/3afirewall/linux-loongson-release/vmlinux
(no debugging symbols found)
```

```
[New Thread 1]
0xffffffff802053d4 in cpu_idle ()
(gdb) info threads
[New Thread 2]
[New Thread 3]
[New Thread 4]
  4 Thread 4 0xffffffff80205414 in cpu_idle ()
  3 Thread 3 0xffffffff802053d4 in cpu_idle ()
  2 Thread 2 0xffffffff80205414 in cpu_idle ()
* 1 Thread 1 0xffffffff802053d4 in cpu_idle ()
```

- 可以利用gdb的monitor命令从gdb端在ejtag-debug上执行命令

```
gdb)monitor setconfig
gdb)monitor dl 0xffffffffbfc00000 10
```

- 每个处理器是一个线程，info threads列出处理器信息
- detach 断开gdb remote连接
- 目前对于多cpu，进入gdb的时候只停止当前的cpu,info threads会停止所有的cpu。
- b,hb,watch,rwatch,awatch等设置断点

4 ejtag硬件

4.1 并口电缆

访问并口可以直接读取pc的io地址，或者打开并口设置/dev/parport0,通过ioctl来访问端口。
并口的pin2-pin9对应数据0-7，作为输出，11,13,15可以作为输入。

两个config和并口配置有关:
pp.pins : 0x00421730:pp pins {23..0}={rst,tclk,tms,tdo,tdi,trst}: 0x401763, 0x421730
pp.ppdev : 0x00000000:pp use /dev/parport0

4.2 usb电缆

- usb电缆 包括:ezusb芯片,FPGA
- FPGA内部有一个mips处理器和一个完成一次ejtag寄存器访问的状态机,一个使用4096的fifo.
- mips处理器是李伟写的mips789 http://opencores.org/project,mips789
- usb采用bulk传输
- 除了实现对ejtag的tap寄存器访问外，还对上载和下载作了特别的优化,速度可以达到1MB/s。

4.2.1 windows下安装

- 龙芯ejtag在windows下使用需要安装驱动，如果windows下没有安装驱动执行ejtag_debug_usb.exe会提示缺少libusb库，或者找不到设备。驱动按照过程如下:

  1. 首先将usb电缆插入pc机的usb口，这个时候windows会检测出未知的usb设备插入提示按照驱动
  2. 选择手动安装，并指定安装目录为ejtag-debug/driver目录,然后下一步来自动安装驱动
  3. 还可以在设备管理器里面找为未安装驱动的usb设备，vid: 2961,pid: 6688，然后指定按照ejtag-debug/driver目录里面的驱动即可。

  驱动安装完后，直接双击ejtag_debug_usb.exe执行就可以了。

- ejtag程序的一些命令会调到脚本，需要安装perl。可以下载active perl或者strawberry perl都可以http://www.perl.org/get.html#win32.
- 如果在cygwin下执行，需要将ejtag-debug里面的cygwin1.dll删除，否则程序会自动退出去。

4.2.2 Linux下安装

在linux下不需要安装驱动，直接以超级用户权限执行ejtag_debug_usb.exe即可。

4.2.3 ejtag快速下载的原理

下面简单解析一下put命令的实现原理

1. 将一小段汇编程序上载到ddr内存里面，并跳的ddr中执行，程序是循环从dmseg地址0xff20fe00中取数据写入ddr内存中,最后跳到0xff200200.
2. 当处理器访问dmseg的时候，处理器会等待pc机端响应写入data寄存器，并清pracc,这里是用fpga来作这个工作
3. pc机软件将上载的数据和大小信息采用libusb库bulk传输给仿真器
4. 仿真器程序读出usb数据,将ir,dri和是否需要读dro等信息写入fifo,直到数据传完。
5. pc机usb传输完后,等待pracc并且地址为0xff200200,这样快速下载就完成了。

FPGA上mips处理器运行的快速下载代码

```
void fast_wr_jtag( u32_t data)
{
    u32_t addr=JTAGC_DIN_BASE_ADDR ;
   addr|=0x40;
    while(is_jtag_fifo_full());
    *(volatile u32_t*)(addr)=data ;

}

void do_opcode_12(void)
{
    u32_t i,cntr ;
    u32_t dr ;

     cntr=USB_RD();
     i=JTAGC_DIN_BASE_ADDR + 0x70;
      while(is_jtag_fifo_full());
     *(volatile u32_t*)(i)=cntr;

    cntr=USB_RD();
    cntr|=USB_RD()<<16 ;
    for(i=0;i<cntr;++i)
    {
        dr=USB_RD();
        dr|=USB_RD()<<16 ;
        fast_wr_jtag(dr);
    }
    i=JTAGC_DIN_BASE_ADDR + 0x70;
      while(is_jtag_fifo_full());
     *(volatile u32_t*)(i)=0;

}
```